УДК 004.657

# ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ БД

**Кучмистий О.Г., Нестеров М.В., Скарга-Бандурова І.С.**

# RESEARCH METHODS OF DATABASE PERFORMANCE OPTIMIZATION

**Kuchmysyi O.H., Nesterov M.V., Skarga-Bandurova I.S.**

*Стаття присвячена вивченню методів оптимізації баз даних у вже існуючих системах. Це складне завдання, тому що, в кінцевому рахунку, вимагає повне розуміння системи в яку входить БД. В окремих випадках для виконання локальної оптимізації досить знати частину системи, проте щоб зробити її більш оптимальною, потрібно розбиратися в тому, як вона влаштована. У цій статті будуть розглянуті різні способи оптимізації MySQL і представлені деякі приклади її виконання. Не слід забувати, однак, що завжди можна знайти деякі додаткові можливості і зробити систему ще швидше.*

***Ключові слова:*** *база даних, оптимізація, SQL запити, підвищення продуктивності існуючих систем.*

**Вступ.** Стрімко зростаючий попит на збір даних та динамічний розвиток бізнесу, призвели до появи проблем, що постійно з'являються на рівні функціонування баз даних. Бази даних використовуються в різноманітних областях застосування і пропонують зберігання складних, як правило, багатовимірних об'єктів. Велика кількість зібраних даних є лише однією проблемою із існуючих. Не менш важливі фактори це збільшення кількості користувачів, розвиток системи шляхом додавання нової функціональності, інтеграція з зовнішніми системами, проблеми з підтримкою серверів, неправильно спроектована схема бази даних и т.п.

Оскільки великі виробничі системи, як правило, мають дуже тривалий життєвий цикл і кількість їх компонентів дуже часто підраховується тисячами, не так рідко зустрічаються додаткові помилки. Велике значення має виявлення таких проблеми та їх своєчасне вирішення.

**Аналіз літератури.** Під час вивчення матеріалу з інших джерел було виявлено ряд особливостей оптимізації роботи баз даних. Найбільш поширена проблема знаходиться на рівні проектування моделі бази даних [1]. Помилки під час цього етапу чи зміни в результаті розробки системи, дуже часто виявляють недоліки моделі [2]. Інший дуже важливий аспект це невірна структура даних, занадто велика індексація стовпців [6]. Дуже висока динаміка та зростання бази даних в поєднанні з багатьма іншими показниками можуть помітно впливати на виконання інструкцій INSERT, UPDATE і DELETE [6]. Деякі проблеми продуктивності виникають через недбале використання тригерів бази даних [8]. Це потужний механізм, який дозволяє автоматизувати процеси. Однак це також може спричинити серйозні проблеми із базою даних, які досить важко виявити в існуючій системі [3].

**Метою цієї статті** є опис проблем і запропонування можливих підходів до їх вирішення. Ці аспекти описані в контексті реальної існуючої системи, побудованої на базі MySQL Server. Однак варто зазначити, що розглянуті механізми, проаналізовані в роботі, можна застосувати на будь-якому сервері баз даних.

**Основна частина**

**Низька продуктивність запитів.** Одна з проблем, що найбільш часто зустрічається в робочих системах - повільна робота у місцях функціонування базових операції. Однією з таких операцій є пошук. Цей крок зазвичай ініціює бізнес-процес. Під час тестів, які зазвичай проводяться на невеликій базі даних, ця проблема не розкривається і відповідно запобіжні заходи не виправлять цю ситуацію.

Розглянемо процедуру пошуку клієнтів у базі даних. З розвитком системи кількість клієнтів в базі даних значно зросла, що вплинуло на час, необхідний для пошуку певного запису.

Наприклад, ми маємо базу даних, в якій зберігається приблизно 1 000 000 даних про клієнтів. Ідентифікація клієнта може здійснюватися за допомогою імені або чисел. Проблема занадто довгих запитів була виявлена під час пошуку за ідентифікаційними номерами. Таблиця 1 описує структуру таблиці «CLIENTS» в базі даних.

Таблиця 1

**Структура таблиці CLIENTS**

| Назва колонки | Тип колонки | Обмеження |
|---|---|---|
| ID | NUMBER(10) | Первинний ключ |
| NAME | VARCHAR(30) | - |
| SURNAME | VARCHAR(30) | - |
| AGE | NUMBER(3) | - |
| ID_DOC_NO | NUMBER(20) | Зовнішній ключ що посилається на DOCUMENTS |

Конструкція SELECT для пошуку по таблиці CLIENTS може бути записана наступним чином:

```
SELECT * FROM CLIENTS WHERE ID = :CLIENT_ID;
```

Відповідно план запитів, представлений на рис. 1, та статистичні запити були зібрані в першому рядку табл. 2.

Як можна помітити, план запиту не є оптимальним, тому що для кожного запиту потрібен двигун бази даних для повного сканування таблиці.

Це впливає на велику кількість дискових зчитувань і на використовуванні буферу. Варто зазначити, що незважаючи на те, що час виконання не довгий, отримані статистичні дані не є задовільними, оскільки слід враховувати кількість запитів на день та одночасні тривалі сесії. Крім того, зростання кількості клієнтів призведе до деградації показників.

Таблиця 2

**Статистика запитів для таблиці CLIENTS**

| Тип запиту | Час (сек.) | Зчитування диску | Використання буферу |
|---|---|---|---|
| Без використання індексів | 1.8 | 64 630 | 128 771 |
| З індексами | 0.01 | 86 | 12 |

Для покращення ефективності пошуку єдиним можливим рішенням є встановлення індексів в стовпці, на якому виконується пошук, за допомогою наступного синтаксису:

```
CREATE INDEX CLIENTS_INDX1 ON CLIENTS(NAME);
```

Це рішення не призводить до зміни логіки застосування. Побічними ефектами створення індексів є: додаткове місце на диску та збільшення часу модифікації та додавання нових елементів. Проте завантаження нових даних в систему, як правило, виконується у фоновому режимі, а модифікація їх відбувається досить рідко. Цей підхід не викликає помітного або різкого падіння продуктивності для інших операцій.

План запиту для пошуку в таблиці CLIENTS з використанням визначеного індексу був змінений оптимізатором, а кількість запитів була зменшена в 70 разів. Середня тривалість запитів після введення індексу для незалежних операцій пошуку показана у другому рядку таблиці 2.

Оптимізація запитів шляхом побудови індексів призвела до зменшення в 700 разів читання буфера для одержання однакових результатів. Єдиним побічним ефектом додавання нового індексу був дисковий простір - 204 Мб.

**Низька продуктивність join – запитів.** Інше питання, що розглядається як проблема трапляється при використанні реляційної системи бази даних. Це об'єднання двох або більше великих таблиць.

Розглянемо звіт про продажі, згрупованих за віком клієнтів. Він базується на таблицях «CLIENTS» (див. таблицю 1) та «SALES_HISTORY» (див. таблицю 3), що містить близько 4 000 000 записів. Приєднана операція базується на колонці «CLIENT_ID», яка вказується в таблиці «CLIENTS». Звіт зазвичай створюється для віку від 15 років. Під час виконання запиту було помічено що на обробку потрібно забагато часу, і тому необхідно провести оптимізацію.

Таблиця 3

**Структура таблиці SALES_HISTORY**

| Назва колонки | Тип колонки | Обмеження |
|---|---|---|
| ID | NUMBER(10) | Первинний ключ |
| SALE_DATE | DATE | - |
| CLIENT_ID | NUMBER (10) | Зовнішній ключ що посилається на CLIENTS |
| TOTAL_AMOUNT | NUMBER (10,4) | - |

Щоб прискорити операцію приєднання, можна зменшити вхідні параметри звіту. [2] Тобто змінити нижню та верхню межу віку або часові рамки. Такий підхід змінює функціональність системи і тому має розглядатися як крайній засіб. Більше того, це не подолає проблему, а просто зменшує вплив факторів, які її формують.

Для підвищення ефективності запиту аналізуються характеристики даних, які зазвичай вибираються для запиту. Було встановлено, що вік розподілу клієнтів зазвичай є однорідним і становить від 20 до 40 років. Тому можна припустити, що у звіті використовується понад 50% відсотків таблиці «CLIENTS» для визначення діапазону клієнтів. З огляду на характеристику даних, план запитів повинен бути проаналізований, щоб визначити, чи є він оптимальним чи ні.

```
SELECT * FROM CLIENTS CL, SALES_HISTORY SH WHERE
CL.CLIENT_ID = SH.CLIENT_ID AND CL.AGE >= 22 AND
CL.AGE <= 36;
```

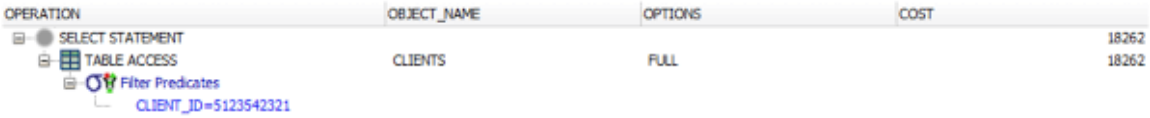


Рис. 1. План запиту для конструкції SELECT без використання індексів [1]

Таблиця 3

**Результат виконання операції EXPLAIN**

| ID | Select type | Table | Type | Possible keys | Key | Key len | Ref | Rows | Extra |
|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | CL | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
| 1 | SIMPLE | SH | ALL | NULL | NULL | NULL | NULL | 1 | Using where; Using join buffer (flat, BNL join) |

Для більш повного аналізу пропонується розглянути результати виконання операції EXPLAIN. Ця операція використовується в різних реляційних базах даних і містить інформацію про те,

Перший рядок таблиці 4 відображає час обробки запитів, зчитування диску та використання буфера. як SQL двигун виконує запит.

Таблиця 4

**Статистика запиту для з'єднання таблиць CLIENTS та SALES_HISTORY**

| Тип запиту | Час, сек | Зчитування диску | Використ. буферу | Вартість запиту |
|---|---|---|---|---|
| Без індексів | 240 | 518 485 | 6 311 813 | 70 345 256 |
| З індексами | 120 | 165 061 | 1 485 492 | 32 315 199 |
| З HASH індексом | 33 | 223 673 | 165 196 | 186 981 |

Давайте розглянемо створення індексу таблиці SALES_HISTORY, щоб запобігти повному доступу до таблиці. Для цього потрібно впевнитися що CLIENT_ID є унікальним і зв'язаний з таблицею CLIENT та атрибутом ID. Створення індексу зменшило вартість і час запиту, а також зчитування диску та використання буферу, але вони залишаються дуже високими. Це пов'язано з специфікою даних, оскільки це приєднання відноситься до більш ніж 50% даних. Крім того, індекс займає 740 Мб дискового простору, і тому будь-які оновлення можуть бути дуже помітними.

Розглядаючи план запитів та характеристики даних, на яких він працює, ми можемо констатувати, що проблема полягає в об'єднанні вкладеного циклу. Запит виконується на великих обсягах даних, і тому приєднання по хешу виглядає більш оптимальним. Щоб оптимізувати такий запит, було використано HASH індекс, при цьому вартість запиту значно знизилася. Треба відмітити що використання такого типу індексів можливе тільки для наступних типів двигуна БД: MEMORY/HEAP, NDB.

Як бачимо, незважаючи на показники, двигун бази даних вибрав повну перевірку обох таблиць [4]. Це найбільш сприятливе, оскільки в більшості випадків ми використовуємо більше 50% даних, тому більша частина таблиці завантажується в пам'ять з блоків даних. Якщо оптимізатор використовував індекс, йому потрібно буде виконати повне сканування, а потім передавати дані за рядком. Як результат, вся таблиця буде завантажена в пам'ять, а час буде значно збільшуватися [2]. Отже, індекс CLIENT_ID, визначений для таблиці SALES_HISTORY, непотрібний і може бути видалений.

Порівняно з варіантом з індексом, на 1/3 більше загальних дискових зчитувань, але кількість використання буферу зменшується в 10 разів, що сприяє збільшенню часу виконання запиту. З точки зору бізнесу, рішення не викликає жодних ризиків, оскільки в логічному слою запити не були змінені.

**Несподіване зниження продуктивності.** Проблема періодичного падіння продуктивності дуже поширена в контексті існуючих систем. Вони можуть відрізнятися, тому важливо точно визначити, яка операція в базі даних є проблемною. Експериментальні дослідження показали який запит є проблемним. Процедура GET_QTY відповідальна за тимчасові затримки. Процедурний кодекс виглядає таким чином:

```
CREATE OR REPLACE PROCEDURE GET_QTY(store_number IN
NUMBER, product IN NUMBER, qty OUT NUMBER)
AS
BEGIN
        SELECT qty INTO qty FROM STORE_ITEM
        WHERE store_id = store_number AND
product_id = product;
END;
```

Процедура виконує лише один запит у таблиці STORE_ITEM. Структура таблиці представлена в таблиці 5. Було створено індекс STORE_ITEM_I1. Зміст таблиці містить близько 600 000 записів даних.

Таблиця 5

**Структура таблиці STORE_ITEM**

| Назва колонки | Тип колонки | Обмеження |
|---|---|---|
| STORE_ID | NUMBER(10) | Унікальний індекс створений для STORE_ITEM_I1 |
| PRODUCT_ID | NUMBER(20) | Зовнішній ключ, що посилається на табл. RODUCTS |
| QTY | NUMBER(30) | - |

Аналіз проблеми, виконаної на момент дослідження показав, що план запиту був виконаний несприятливим чином. У цьому випадку оптимізатор використовував індекс, який був ефективним для невеликих частин табличних даних. На жаль, інший пошук для більшої вибірки також використовував індекс, оскільки запит вже зберігався в кеш-пам'яті бази даних. Якщо запит не був розміщений у кеш-пам'яті бази даних, а план був встановлений з нуля, існували великі шанси, що план буде правильно налаштовано, і сама операція триватиме довго. Можливо, при проектуванні цього розділу системи передбачалося, що розподіл продуктів у магазинах рівномірно розподілений і характеризується високою селективністю, і тому були створені індекси STORE_ID та STORE_ITEM_I1. Однак, це показало, що припущення було неправильним і, в деяких випадках, можуть спричинити більше проблем, ніж переваг.

Ми можемо розглянути три різні способи вирішення цієї проблеми:

- видалити проблемний індекс STORE_ITEM_I1;
- примусити робити повне сканування таблиці для кожного оператора вибору;
- створення нового індексу, який буде більш придатним для всіх виділених заяв.

Перше з наведених вище рішень є найбільш трудомістким та небезпечним у контексті існуючої системи, оскільки видалення індексу в складних системах може спричинити проблеми продуктивності та інтеграції в інших частинах системи. Оскільки виробничі системи, як правило, є багатофункціональні, і їх розробка та підтримка потребує багатьох людей, цей підхід слід відхилити [1].

Примусове сканування таблиці не веде до комерційного ризику. Це рішення повністю виключить проблему продуктивності. Проте повне сканування збільшить час вибору з меншого набору даних. Тим не менш, час підйому від 0,003 до 0,26 секунди буде незначним [5]. Навіть якщо число користувачів збільшиться, двигун MySQL досить розумний, щоб кешувати подібні, досить навантажені, запити.

Найкращим варіантом може бути створення індексу, який включатиме стовпці STORE_ID і PRODUCT_ID [1].

Можна помітити, що індекси мають помітний вплив на час виконання, а також зчитування диска та використання буферу. Вартість запиту значно зменшилася через те, що за допомогою індексу двигун бази даних точно знає, які блоки даних повинні посилатись на завантаження отриманих даних. Тому з диска відсутні надмірні зчитування. Однак, враховуючи те, що пошукові операції виконуються багаторазово в системі, можна стверджувати, що завантаження всієї таблиці також є корисним. Це пов'язано з тим, що кожен наступний пошук, скоріш за все, скористається таблицею, яка вже завантажена в буфер, або, в крайньому випадку, зчитає лише невелику частину таблиці з диска.

**Пакетна обробка даних.** Пакетна обробка - це дуже поширена проблема в робочих системах. Вона зосереджується на завантаженні та, в деяких випадках, обробці вхідних даних з зовнішніх систем. Обсяги даних часто дуже великі, і їх завантаження або обробка повинні бути закінчені протягом короткого часу.

Розглянемо процес, який завантажує зовнішні дані, доповнені певними атрибутами, на основі цільової системи. Оскільки обсяг вхідних даних та таблиці великий, потрібна оптимізація. Варто зазначити, що дані не є критичними з точки зору бізнесу. Крім того, невелика зміна логіки прийнятна, оскільки дані перевіряються після завантаження.

У системі є 1 000 000 найменувань і 10 місць. Кожен елемент є в ієрархії і має батька. Багато предметів може мати одного й того ж батька. Структурні таблиці представлені у таблицях 7, 8 та 9. Щомісяця завантажується файл, в якому входять всі елементи з кількістю одиниць, що продаються з кожного місця. Ці елементи призначені для будь-якого батька, і їх необхідно об'єднати з відповідними записами з таблиці ITEMS. Неможливо виконати одну операцію UPDATE, оскільки ця система отримує щомісяця близько 4 000 000 даних.

Таблиця 7

**Структура таблиці ITEMS**

| Назва колонки | Тип колонки |
|---|---|
| ITEM_ID | NUMBER(20) |
| ITEM_PARENT | NUMBER(20) |
| DESCRIPTION | VARCHAR(200) |

Таблиця 8

**Структура таблиці LOCATIONS**

| Назва колонки | Тип колонки |
|---|---|
| ITEM_ID | NUMBER(20) |
| ITEM_PARENT | NUMBER(20) |
| DESCRIPTION | VARCHAR(200) |

Запит, що оновлює дані, можна записати таким чином:

```
UPDATE EXTERNAL_SALE scp SET ITEM_PARENT = (SELECT
ITEM_PARENT FROM ITEMS I WHERE I.ITEM_ID =
SCP.ITEM_ID)
```

Таблиця 9

**Структура таблиці EXTERNAL_SALE**

| Назва колонки | Тип колонки |
|---|---|
| ITEM_ID | NUMBER(20) |
| LOCATION | NUMBER(20) |
| ITEM_PARENT | NUMBER(20) |
| QTY | NUMBER(3) |

Запит, виконаний для повного діапазону даних 40 000 000, не закінчився протягом 70 хвилин. Подальше очікування виконання запиту марно і не може бути продовжено. Крім того, він несе ризик помилки бази даних через відсутність табличного простору.

Одне з рішень полягає в тому, щоб розділити весь діапазон даних для менших підмножин і повторити одну й ту ж дію для кожного з них. Вся операція була розділена на частини з будівництва так званих BULK COLLECT. Крім того, пропозиція FORALL використовується для того, щоб виконати оновлення всієї підмножини, і COMMIT додано для запобігання проблемам з відмовою табличного простору. [2] Код програмування був визначений таким чином:

```
DECLARE
        CNT NUMBER := 1;
        CURSOR SCOPE_CUR IS
        (SELECT ROWID, ITEM_ID FROM EXTERNAL_SALE);
        TYPE rowid_type IS TABLE OF rowid;
        ROWID_TAB ROWID_TYPE;
        TYPE ITEM_TYPE IS TABLE OF
        EXTERNAL_SALE.ITEM_ID%TYPE INDEX BY
PLS_INTEGER;
        ITEM_TAB ITEM_TYPE;
BEGIN
        OPEN SCOPE_CUR;
        LOOP
        FETCH SCOPE_CUR BULK COLLECT
        INTO ROWID_TAB, ITEM_TAB LIMIT 100000;
        FORALL I IN ROWID_TAB.FIRST..ROWID_TAB.LAST
        UPDATE external_sale scp SET ITEM_PARENT =
        (SELECT ITEM_PARENT FROM ITEMS I
        WHERE I.ITEM_ID = item_tab(i))
        WHERE SCP.ROWID = ROWID_TAB(I);
        IF(MOD(CNT, 10) = 0) THEN
        COMMIT;
        END IF;
        CNT := CNT + 1;
        EXIT WHEN ROWID_TAB.COUNT = 0;
        END LOOP;
        COMMIT;
END;
```

Таблиця 10 показує статистику оновлення даних, що виконуються за допомогою єдиного виразу SQL та BULK COLLECT.

Таблиця 10

**Статистика UPDATE запиту**

| Тип | Час виконання | Зчитування диску | Використання буферу |
|---|---|---|---|
| UPDATE | 4200 (незакінч.) | - | - |
| BULK COLLECT | 960 | 460 596 | 126 939 278 |

Можна помітити, переглядаючи табл. 10, що оптимізована команда BULK COLLECT, читає необхідні дані з диску, а потім обробляє їх за допомогою буферу [1].

**Висновок.** Розвиток існуючих інформаційних систем передбачає необхідність розробки та вдосконалення механізмів оптимізації баз даних. Методи оптимізації, описані в роботі, є достатніми для визначення та вирішення дефіциту продуктивності. Доступні оптимізаційні механізми допомагають мінімізувати ризики, пов'язані з тим, що система працює у виробничих середовищах. Підхід до кожної проблеми має відповідати загальним правилам, але також слід приймати окремі міркування та конкретний аналіз ситуації. Єдині процедури оптимізації всіх проблем можуть призвести до їх неправильної класифікації, а отже до неправильного методу їх вирішення. Контекст існуючої системи викликає суттєві обмеження щодо доступних методів оптимізації баз даних. Бізнес ризики, пов'язані з підривом стабільності робочої системи, є важливим фактором у остаточному виборі методів. Як правило, оптимізація спрямована на зменшення кількості конкретних ресурсів, необхідних для отримання результату, що впливає на час виконання для SQL-запитів або частин коду.

**Література**


1. Powell, G., Oracle Performance Tuning for 10gR2, Elsevier Inc., 2007.
2. Koper, R., Analysis of the database optimization methods in the context of existing systems, Master's thesis, Lodz University of Technology, 2015.
3. David Kroenke, David Auer. Database Concepts (3rd Edition). – M.: , 2007.
4. Michael Widenius. MySQL Reference Manual. – M.: , 2002.
5. Hugh E Williams. Web Database Applications with PHP & MySQL. – M.: , 2002.
6. Darl Kuhn, Sam R. Alapati and Bill Padfield, Expert Oracle Indexing and Access Paths, Apress Inc., 2016.
7. Ibrar Ahmed, Gregory Smith, Enrico P., PostgreSQL 10 High Performance, Amazon Digital Services LLC, 2017.
8. Benjamin Nevarez, High Performance SQL Server, Apress Inc., 2016.


**References**


1. Powell, G., Oracle Performance Tuning for 10gR2, Elsevier Inc., 2007.
2. Koper, R., Analysis of the database optimization methods in the context of existing systems, Master's thesis, Lodz University of Technology, 2015.
3. David Kroenke, David Auer. Database Concepts (3rd Edition). – M.: , 2007.
4. Michael Widenius. MySQL Reference Manual. – M.: , 2002.
5. Hugh E Williams. Web Database Applications with PHP & MySQL. – M.: , 2002.
6. Darl Kuhn, Sam R. Alapati and Bill Padfield, Expert Oracle Indexing and Access Paths, Apress Inc., 2016.
7. Ibrar Ahmed, Gregory Smith, Enrico P., PostgreSQL 10 High Performance, Amazon Digital Services LLC, 2017.
8. Benjamin Nevarez, High Performance SQL Server, Apress Inc., 2016.

**Кучмистый А.Г., Нестеров М.В., Скарга-Бандурова И.С. Исследование методов оптимизации производительности БД**

*Статья посвящена изучению методов оптимизации баз данных в уже существующих системах. Это сложная задача, потому что, в конечном счете, требует понимания системы в целом. В отдельных случаях для выполнения локальной оптимизации достаточно знать часть системы или приложения, однако чтобы сделать систему более оптимизированной, нужно разбираться в том, как она устроена. В этом материале будут рассмотрены различные способы оптимизации MySQL и представлены некоторые примеры ее выполнения. Не следует забывать, однако, что всегда можно найти некоторые дополнительные возможности и сделать систему еще быстрее.*

***Ключевые слова:*** *база данных, оптимизации, SQL запросы, повышение производительности существующих систем.*

**Kuchmystyi O.H., Nesterov M.V., Skarga-Bandurova I.S., Research methods of database performance optimization**

*The article is devoted to the study of database optimization methods in existing systems. This is a difficult task, because, it requires an understanding of the system as a whole. In some cases, to perform local optimization is enough to know part of the system or application, but to make the system more optimized, you need to understand how it works. This material will look at various ways to optimize MySQL and present some examples of its implementation. We should not forget, however, that you can always find some additional features and make the system even faster.*

***Key words:*** *database, optimizations, SQL queries, improving the performance of existing systems.*

**Кучмистий Олександр Георгійович,** магістрант кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, email: alexandrk95@gmail.com

**Нестеров Максим Володимирович,** ст. викл. кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, email: mnesterov@snu.edu.ua

**Скарга-Бандурова Інна Сергіївна,** д.т.н., зав. кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету імені Володимира Даля, e-mail: skarga-bandurova@snu.edu.ua

*Рецензент:* д.т.н., доц. ***Горбунов М.І.***

Стаття подана 29.10.2018